Shared Memory Programming: Now share objects into shared memory in the fastest way and with fewer steps

Applies to:

SAP NetWeaver Application Server release equal or higher than 6.40

Summary

This article deals about the technique of sharing the class instances and anonymous data objects into shared memory. This has got many advantages over the earlier memory sharing techniques in ABAP in terms of flexibility, speed and performance volumes for sharing data from within ABAP programs.

Author(s): Priti Mulchandani

Company: SAP Labs India Pvt. Ltd.

Created on: 27 December 2006

Author Bio

Priti is working as development specialist in SAP Labs India Pvt. Ltd. from last 2 years, with 3 and half years of total SAP experience. She has mostly worked on different customer specific projects related to area of IS-AFS and has keen interest on anything new in ABAP.

Table of Contents

Why Shared Memory and History

Performance, Long run times, Avoiding database tables for sharing few changing contents from one program to another was all pointing to one thing – Share your data through Memory .

Well we had few techniques before, to start with “EXPORT FROM” and “IMPORT TO” ABAP memory – but within current main session. Then came; SPA/GPA mechanism. We were able to share data for few fields across External Sessions via SAP memory. Of course this was not really comfortable, we wanted more- as and when our program sizes grew, the requirement of data for sharing across sessions and across users was a need in a really speedy way. So SAP made it possible by allowing us to store huge data and complex structures through help of “Cluster Tables”. All these techniques internally copy cluster data from any kind of memory to our own roll area or context of the program, which takes fair amount of time in case of huge data and many users.

Sometimes we even felt database tables will be faster and easier way of programming rather than memory sharing. So can’t we write simple program with good performance whenever we want to share data via memory access. Well the answer now is “Yes”. We have now something called “Shared Object Memory”. So; is this kind of memory really new? “Not Really!” We already had something called “Shared Memory” at application server level. The difference lies in the word “Object” ;-). But it meant a lot and like everything now being converted to OO, Memory programming also got adapted to it in ABAP. The idea is, we can store instances and hence the data in the extended place known as “Shared Object Memory”. Actually all of us have already seen shared memory access but we never realized that ABAP runtime uses it for its own system programs. For example, we could see the best use of shared memory in Workbench – SE80, when we shift from one object to other, when we double click on some db table name to switch to SE11, when we debug using new ABAP debugger to see the contents at runtime, when we double click on some FM/subroutine and come back to original place…. So this kind of memory was always there – “Everywhere”.

Some Basics about Shared Object Memory

The idea is that **SOM (Shared object memory**) is available at application server level hence it can be accessed across transactions and can be shared among the different users in effective way.

We can now come to actual part and its use. At programming level, we will be dealing with One Area class, Root class, one write program, one read program and tool to define SOM. There are few terminologies though, so let’s understand those first:

Transaction SHMA – simple transaction to define shared memory areas and its properties, this is as easy as using SE24 or SE37.

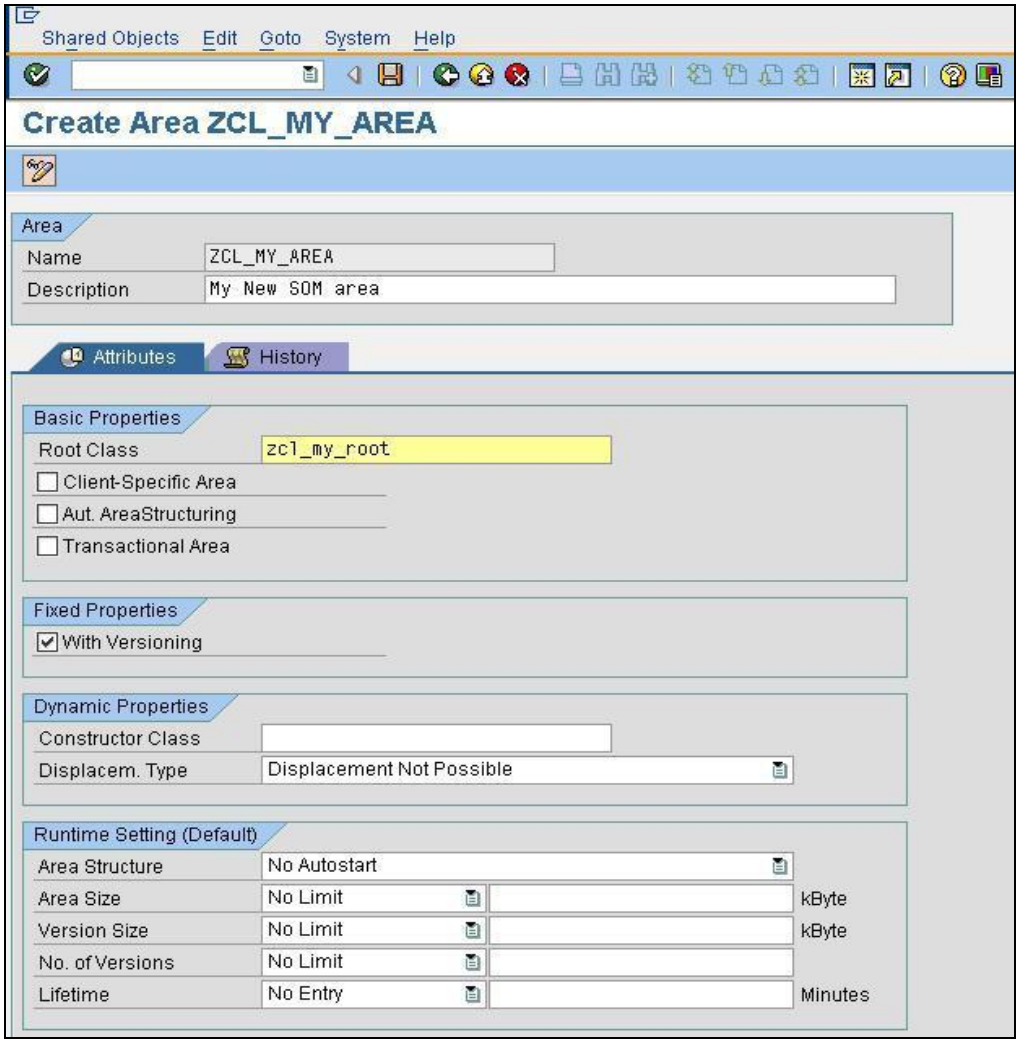

Fig 1: Transaction SHMA – Defining Shared memory Area and its properties

Area – Just think that you have to give a new name and reserve some piece of the SOM for sharing your own object -something like area for building your own house. Technically system differentiates one AREA from other AREA by some properties/characteristics. A class with the same area name gets automatically generated once you define your area in SHMA.

Area Instance – Can be compared with different instances at runtime of any usual class, but truly speaking it's not the OO instance. It is again a sub portion of the defined area, which actually gets filled with data through our programs. We can export multiple real OO instances (multiple sets of data) from our programs to different SOM area instance.

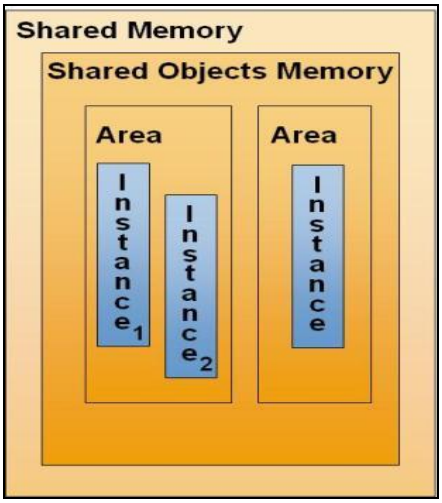

Fig 2: SOM Area instances

Root Class – One of the very important properties of the AREA that we will define using SHMA. Truly this is the class whose instance (mentioned above as real OO instance), we will be storing in SOM from our program. Any access to area instance is always through root instance. This class is same as any other global final class but should necessarily have a property called "Shared Memory Enabled"

Area Instance Version/Versioning – Very unique concept that we didn't have so far. If a reader is reading the contents of particular SOM instance and the writer also wants to write or update new contents in same instance, then a new version gets generated every time. Something like - old, new, latest, being build, expired…. There has to be always one version with which SOM runtime will work and just remember that a reader always reads active/last committed version if there are multiple versions in SOM. Meanwhile if Writer has finished updating and if a new active version is now available, the old version will be automatically garbage collected by SOM runtime.

Locks in SOM – If one writer is writing into same memory area instance, another user should not get access for writing again. Well the beauty is once you start writing in SOM, system will place a change lock, and hence no other writer can get the change access on the whole application server for same SOM instance. On the other hand, Readers lock remains in effect till current session. Thus with the help of above property, the lock mechanism assures us to get the access to the active/most recent version every time.

Transactional commit – It's difficult to assure the proper synchronization between memory and database contents in our programs sometimes, so this assures that only when you do db commit, the related content will be written to SOM.

There are few other properties such as

Propagation to propagate the changes of SOM from one application server to another of same SAP system

Automatic preloads when someone has cleared/refreshed the contents in SOM (which is directly possible though single button or through different methods of generated area class), but you want to place it again automatically without manually executing your write/export program

Freeing complete or part of memory

Deciding the **Size** of memory area/instances (Also check abap/shared_objects_size_MB profile parameter)

Client separation at memory level which is something similar to specifying "client" in select query.

This is basically done once at design time to define our areas and now it's the time to move to simple programming to export and import something in these areas.

How to Use SOM – Simple Export and Import Programs

There is one important object called ARE AHNDLE which is a key to write or read objects in SOM. This is an instance of AREA class and allows us to do many more things with the help of generated area class's methods. AREA class is kind of wrapper which hides technicalities of how actually the OO-instances (root class's instances) are written inside SOM. In the place of Export and Import statements, we will be using write and read instance methods of the area class. This is kind of entry point from our program to root object in SOM instances.

So for sharing data through our code, we will be using two instances AREA HANDLE and ROOT INSTANCE. Methods of AREA HANDLE will be used to place ROOT INSTANCE inside shared memory.

Following well known ABAP statements are upgraded in the context of SOM–

DEFINE CLASS *zcl_my_root …*SHARED MEMORY ENABLED – With this you say that your class is going to also take place in SOM. Check out properties tab for the same in SE24 while defining root class.

CRETAE OBJECT *myRoot* AREA HANDLE *myShmHandle*

CREATE DATA *dref* AREA HANDLE *myShmHandle*

So we should now realize the fact that we not only store object reference but data references also (from release 7.0).

Our piece of Export code or WRITE program should look like this:

```
\\* Define Area handle and Root instance
Data : myShmHandle TYPE REF TO zcl_my_area,  "zcl_my_area is same as area name
       myRoot      TYPE REF TO zcl_my_root.

*Get area class instance or area handle and use handle's ATTACH_FOR_WRITE
*method to export your data in SOM.
myShmHandle = zcl_my_area=>attach_for_write( ).“You can specify instance name, here DEFAULT is used

* Start filling the contents of shared memory enabled root class into SOM instance
CREATE OBJECT myRoot AREA HANDLE myShmHandle.
myRoot->name = `My first area instance`. "name is simple public attribute in root class
APPEND `Harry` TO myRoot->itab. "itab defined as public internal table of type string
* dref is data reference to string. You can even try with dynamic data types
CREATE DATA myRoot->dref AREA HANDLE myShmHandle.
myRoot->dref->* = `Jul-04-2005`.

IF ...
* tell area handle who is going to work as root- don't miss this, Root is always stored as attribute of area class
  myShmHandle->set_root(  myRoot ).
* Finally put the root instance (and hence all its attributes/data) in SOM
  myShmHandle->detach_commit( ).
ELSE.
```



```
    myShmHandle->detach_rollback( ).
  ENDIF.
  * After this if you will fill in attributes of root instance w/o above blocks; that won't sit in SOM at all
  ....
  *However if you repeat another similar block, this time with new data for example in root object, a new SOM instance will\
  * get created
  ….
```

Now let's have a look at READ Program – This mostly contains opposite methods of AREA class to import the instances:

```
  \\ DATA:
    myShmHandle TYPE REF TO zcl_my_area,
    myRoot      TYPE REF TO zcl_my_root,
    txt         TYPE string.
  *  This time get the handle through static ATTACH_FOR_READ method of area class
  myShmHandle = zcl_my_area=>attach_for_read( ). "Try catching different exceptions of methods

  *Root instance should already be active and present as writing would have been already done using export pgm.
  myRoot = myShmHandle->root.

  *Now read the contents as you normally do w/o SOM
  READ TABLE myRoot->itab INTO txt INDEX 1.
  WRITE: / txt.

  CONCATENATE myRoot->name
              myRoot->dref->* INTO txt
              SEPARATED BY space.
  WRITE: / txt.
  *Tell system, that you have finished reading, this will remove read lock
  myShmHandle->detach( )."Even if you don't detach, sys will remove at the end of pgm

  ….
```

So with this, we are done with placing our references/data/complex structures in the memory.

But how we will be sure it is really working, questions like "Can I see my memory contents after placing? Can I delete un-needed objects? Can I go and see how many readers are currently reading the object? Or who is the writer, so that I can fine tune my program" always hover around us while dealing with memory programs. And answer to these and many more questions finally lies with the interesting transaction **SHMM** – A powerful monitor tool. Check your areas, Instances, Versions, Locks, and even **Data/attributes of** the instances of root class and try many more things such as delete.

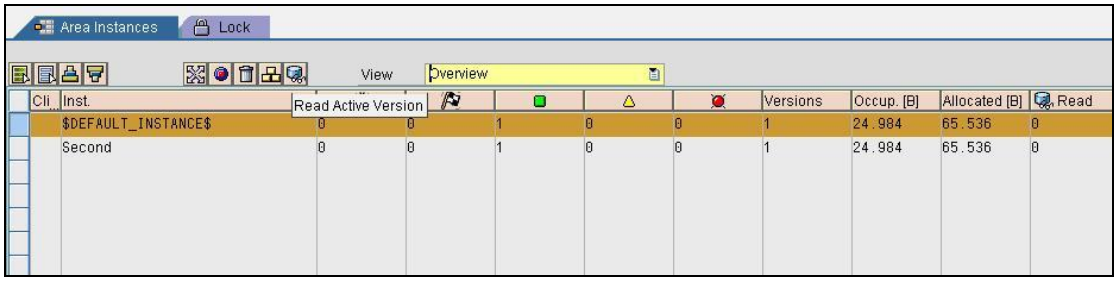

Fig 3: SHMM – View instances and data

Conclusion

To end with, there are few restrictions with SOM programming model. As nothing can be absolutely the best option, there is always a chance of getting something more powerful than this. But it is good to end with interesting true figures:

With several hundreds of users, a control and monitor tol took 16000 ms whereas shared memory program just took 3000 ms. this is because-

-> Without SOM: there was 3 MB memory per user session (50 users = 150 MB)

-> With SOM: 3 MB for all users – now there is direct copy-free access to shared memory i.e. no copy to session memory or roll area

➔ Result Access performance: SOM is 100 times faster at first access in a user session!!

Copyright